Final Report on

Investigating Better Alternatives for MarketMaker Database

Jong S. Lee, Ph.D., Senior Research Scientist
National Center for Super Computing Applications
University of Illinois at Urbana-Champaign

A Report submitted in fulfillment of the requirements for the Market Maker Enhancement Project titled, “Investigating Better Alternatives for MarketMaker Database”

Table of Contents

List of Tables

List of Figures

Investigating Better Alternatives for MarketMaker Database

1 Introduction

MarketMaker has various schemas of data since the system covers a comprehensive list of agricultural products. The same type of data may have different schemas. For example, the business profile for Farmer/Rancher (Figure 1) is very different from the Fishery profile (Figure 2). The issue with this characteristic is raised when users want to search across different types of business profiles. This uniqueness of the MarketMaker data has been solved in the traditional Relational database. However, due to the nature of a relational database, the scalability and the flexibility of adding new types are in question.

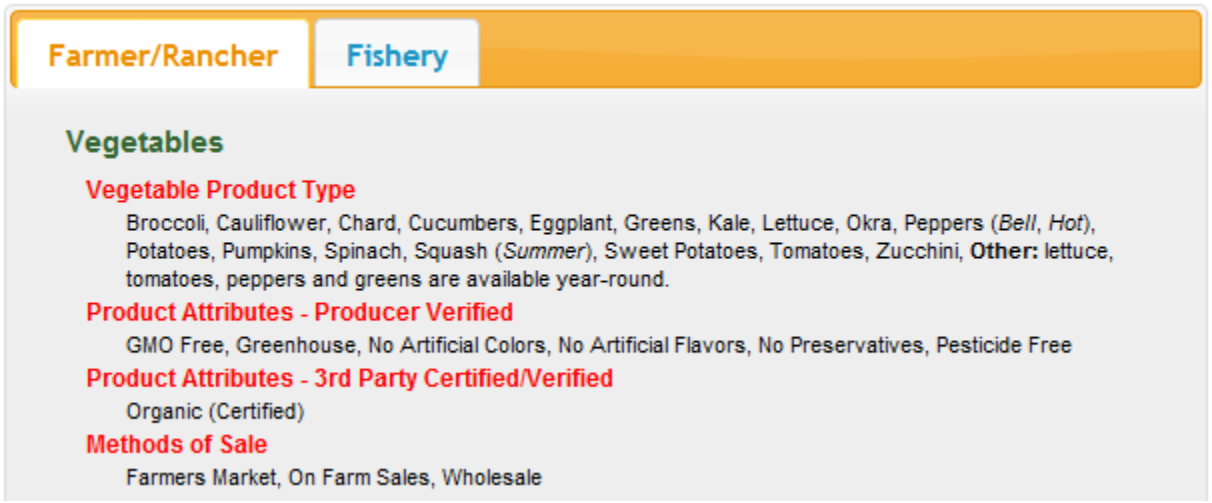


Figure 1 - Farmer/Rancher schema

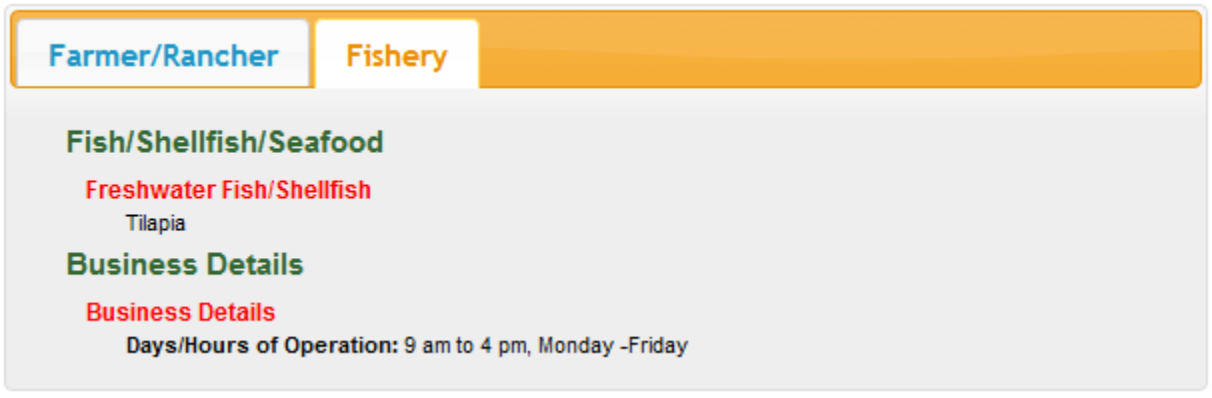


Figure 2 - Fishery schema

This kind of problem has been addressed by the database community and, recently, an alternative called NoSQL data storage has been developed. In order to serve loosely defined (flexible) schema like MarketMaker, the document store (such as MongoDB, CouchDB, etc.) and the column store (such as Apache Cassandra) have been implemented.

Section 2 of this report details our investigation of MarketMaker's current data storage and data structure. Based on the issues identified in the current MarketMaker, new possible alternatives are outlined in Section 3. In Section 4, we implemented and tested the selected technology, MongoDB, with current data in MarketMaker. According to our findings, we provide the recommendation in Section 5.

2 Current MarketMaker Data Storage and Structure

MarketMaker data is currently housed in a traditional relational database management system—Microsoft SQL Server 2008 running in a Window Server environment. The user-submitted and purchased business data is stored here, as are the components used by the web application, to build the site dynamically. The schema of the database is shown at Appendix A. At the heart of the system is the business record. Nearly all searchable aspects of the database are related directly to this one, core piece of the puzzle. Business owners and employees, industry affiliation membership details, account preferences, and login credentials are all tied to this key record. Yet the aspect most important to the success of the site is the profile information associated with a business. Herein lies the true means by which users can find businesses, based on an arbitrary number of products, attributes, product forms, methods of sale, and more.

Currently, the profile structure is contained within a single table in the database (shown at Appendix A). Business' claims to these elements are contained in another table, but the key structure of all of the profiles, attributes, product forms, etc. that can be claimed are kept in a master table. This profile table is constructed as a hierarchy base on the adjacency model. The database informs the application when building forms for capturing the profile data. Additionally, this hierarchy is currently used to build the searching platform. In this way, the taxonomy of profile data is the same for both entering business data and searching for businesses.

2.1 Advantages

There are several advantages of the current model of storing profile information. First, the system is extremely flexible. Changes to the taxonomy are generally minor tasks, and even in the case of adding, subtracting, or rearranging major portions of the taxonomy, edits can be made with relative ease, due to the organic nature of the hierarchy and its central location. Second, because this one table informs both the registration and search portions of the site, a common language of use exists for the entire MarketMaker system and modifications require no editing of the application code. Finally, users are able to conduct searches at any level of granularity, from very general to exceptionally specific, thanks to the hierarchical nature of the data.

2.2 Problems and Issues

Even with such strengths, there are shortcomings associated with this model. Chief among them is the search experience for users. Depending on the audience, being forced to use the registration classification of products, attributes, etc. as a means of search can be frustrating. Keyword searching is the most requested feature of the system. Even so, implementing it is no small challenge since the context of each item in the taxonomy must be considered when parsing and executing a keyword request.

3 Alternative Technology

Problems and Issues identified in the previous section have been addressed by the database community and recently, the alternatives have been developed. It's called NoSQL data storages. In order to solve the types of issues MarketMaker has, the document store (such as MongoDB, CouchDB, etc.) and the column store (such as Apache Cassandra) have been implemented.

In this section, we will review the NoSQL. The document store and column family store are also reviewed in detail.

3.1 NoSQL[1,2]

So far, the relational database has provided a good bridge between the users and the information providing the method of storing and retrieving the data. With the growth of internet users and applications, the amount of data is also growing exponentially with the inflow of new information and digitalization of the old information. The burden of storing and retrieving the data is also increasing. Along with the size and complexity growth of the data in the database, the necessity for higher performance and scaling has also increased requiring a new solution that the typical database application is not able to provide.

NoSQL (represents 'not only SQL') is a rapidly growing database management system that was originally developed for modern web-scale databases in early 2009. Unlike the traditional RDMBS (Relational Database Management System), its data structure does not require the table to build the database and relational model, and it is being developed for next level with the non-relational, distributes, and horizontally scalable database. In NoSQL, it is more important to store and retrieve large amount of data than data structure or the relationship between elements. Currently it is being used by major internet companies such as Google and Amazon mainly for the ability to handle the huge amount of the data, and other advantages like distributed and fault-tolerance architecture, easy replication support, and simple API. There are several types of NoSQL databases:

- Wide Column Store / Column Families: Hadoop, Cassandra, Hypertable, etc
- Document Store: MongoDB, CouchDB, RavenDB, etc
- Key Value / Tuple Store: DynamoDB, Azure Table Storage, etc
- Graph Databases: NeoJ, Infinite Graph, Sones, etc
- Multimodel Databases: OrientDB, ArangoDB, AlchemyDB, etc
- Object Databases: db4o, Versant, Objectivity, etc
- Grid & Cloud Database Solutions: GigaSpaces, Infinispan, Queplix, etc
- XML Databases: Mark Logic Server, EMC Documentum xDB, eXist, etc
- Multidimensional Databases: Globals, Intersystesms Cache
- Multivalue Database: U2, OpenInsight, Reality, etc
- Other NoSQL related databases: IBM Lotus/Domino, eXtremeDB, RDM Embedded, etc
- Unresolved and uncategorized: VMware vFabric GemFire, KirbyBase, Tokutek, etc

[1] http://en.wikipedia.org/wiki/NoSQL
[2] http://nosql-database.org/

Large amount of data management and elastic scaling using distributed architecture are probably the biggest advantages of using NoSQL[3]. In most cases, buying bigger servers (scale up) is often provided as a solution when the database load increases rather than distributing the database load to multiple hosts (scale out) in RDBMS. Scaling which moves the database into multiple hosts or cloud in NoSQL is much easier than RDMBS since it is designed to employ a distributed architecture, letting the system expand horizontally by adding more servers. This increases the system's tolerance of the server failure as well. This horizontal scalability is also effective in big size data management when real-time performance is more important, such as a huge amount of document indexing and serving web pages in high-traffic web sites rather than the consistency than functionality. It also requires less management due to its simpler data models and its automatic data repair and distribution.

In RDBMS, one of the big challenges is a change management, even if the change is very minor. It must be implemented very carefully with plans and time. Since NoSQL does not require any complex or fixed database schema, it provides less data model restrictions and allows the application to store the data with any structure.

3.2 Document Store[4]

Document store is one concept in NoSQL databases. Unlike the RDBMS notion of 'Relations' or 'Tables', document storage is the notion of 'Documents'. In this system, documents are the data that are encapsulated and encoded in standard formats or encodings, including XML, JSON, YAML, and BSON. Modifying the structure of any document is relatively easy by adding and removing the members from the document. Organizing and grouping elements of documents is different from the traditional RDBMS through collections, tags, non-visible metadata, and directory hierarchies. Collections are like tables in a relational database. The main difference between these is that records in the table have the same number of fields but collection allows documents to have totally different fields. Document has a unique key, often a simple string that represents the document and in the database and used for retrieving the data. Another characteristic is that content-based document retrieving using the API or query is possible[5]. Detailed explanation is in the following chapter using MongoDB, one of the popular and widely-used Document Store.

3.2.1 MongoDB[6]

The name MongoDB came from the word "humongous". It is an open source (AGPL)[7], scalable NoSQL database written in C++. The main feature of the MongoDB is document-oriented storage, full index support, replication ability, etc. It uses JSON-style document for its document storage, and allows indexing of any attribute. Network-based mirroring allows horizontal scalability using auto auto-sharding

[3] http://www.techrepublic.com/blog/10things/10-things-you-should-know-about-nosql-databases/1772
[4] http://en.wikipedia.org/wiki/Document-oriented_database
[5] http://slashdot.org/topic/bi/nosql-document-storage-benefits-drawbacks/
[6] http://www.mongodb.org/
[7] http://www.mongodb.org/display/DOCS/Licensing

that allows users to distribute the data across multiple nodes. Files of any size can be stored with less complication to the database stack.

Many features in MongoDB are very similar to the relational databases so it could be a good bridge between the SQL and NoSQL, providing users a good first step to NoSQL and document storage type databases. Its performance speed is very fast and geospatial queries are built in so it does not require any plugins. Table 1 shows the similarity between MongoDB and the typical relational database using SQL terms.

Table 1 - Comparing terms between relational DB and MongoDB

Relational DB term	Mongo term/concept
database	database
table	collection
index	index
row	BSON document
column	BSON field
join	embedding and linking
primary key	_id field
group by	aggregation

In MongoDB, the records are stored as BSON (binary form of JSON) and records within a single collection has a similar concept as a table in a relational database that can have different structures. Here's an example of single record:

```
{
   "_id":ObjectId("4fccbf281168a6aa3c215443"),
   "first_name":"George",
   "last_name":"Washington",
   "address":{
      "street":"504 East Pennsylvania Ave",
      "city":"Champaign",
      "state":"IL"
   }
}
```

This record has attributes of ObjectId, first_name, last_name, and address that have an object with more data inside it. The ObjectId is MongoDB's automatically generated unique ID. These records are called documents and the structure of this document can be modified by adding and removing the members because there is no schema. If a spatial record is added, such as longitude and latitude, the spatial query is also possible.

3.3 Column Family Store[8]

In relational database, a table contains all the data structure (schema). It defines column names and their data types. Each row has an individual set of data defined by the fixed set of columns. Unlike relational database systems that stores data tables as rows of data, column family storage stores sections of columns of data. Column family storage uses the column family that is a tuple (pair) consists of a key-value pair that contains the related data. In key-value pair, the value that is a set of columns is bound to the key. In comparison to relational database, a column family is similar to a 'table' that contains columns and rows, and each key-value tuple is as a 'row'.

In real-life situation, the column family is not completely schema-less, even though it is very flexible, because each column family should contain a single type of data[9]. This system has advantages when the data has very similar items, like library card catalogs, census data, or data warehouses. The next chapter has a more detailed explanation about Column Family Storage database using Apache Cassandra, a widely-used and well-developed implementation.

3.3.1 Apache Cassandra[10]

Cassandra is an open source distributed database management system developed by Apache Software Foundation. It provides linear scalability and fault-tolerance on commodity hardware or cloud infrastructure for handling huge amounts of data and distributing them across multiple commodity servers. Cassandra is a decentralized system that provides multiple nodes in the cluster that are identical with no single points of failure or network bottleneck. Elastic structure provides the convenience of linear scalability as new machines can be added without any interruption, such as downtime or pause of application. Cassandra supports replication across multiple data centers and has an ability of automatic data replication, to multiple nodes for fault-tolerance.

The smallest unit of data in Cassandra is the column that is a tuple containing a name, a value, and a timestamp. The major task in using Cassandra is to define column families. Like other column family storage database systems, each row in Cassandra can have a different set of columns, and the metadata about these columns can be defined by column families. Cassandra has key-value structure where key maps to multiple values are grouped into column families. Even though the Cassandra database creation process fixes the column families, a family can have additional columns at any time.

There are two typical column family design patterns in Cassandra - static and dynamic. A static column family is similar to a relational database table since it uses a relatively static set of column names, and column metadata for each column is predefined. In the following table example, the rows have similar columns, such as name, email, and address, but they don't have to have all of the columns defined.

[8] http://en.wikipedia.org/wiki/Column-oriented_DBMS
[9] http://wiki.toadforcloud.com/index.php/Column_Families_101
[10] http://cassandra.apache.org/

Table 2 - Example of Column family store

row Key	columns…				
gwash	first_name	last_name	email	address	state
	george	washington	gwash@test.org	123 green	IL
jadams	first_name	last_name	email	address	state
	john	adams	jadams@test.org	45 main	TX
tjefferson	first_name	last_name	email		
	thomas	jefferson	tjefferson@test.org		

A dynamic column family can store pre-computed result sets in a single row for efficient data retrieval using Cassandra's ability to use arbitrary application-supplied column names. In the following table, each row is a snapshot of data that contains the users' personal information shown in previous table.

Table 3 - Pre-computed data

row Key	columns…			
gwash	jadams	tjefferson	jmadison	jmonroe
jadams	tjefferson			
tjefferson	jmadison	jmonroe		

3.4 Technology to Investigate Further

In order to store various schemas available in MarketMaker, the document store and the column family store are considered. However, the column family store is less flexible than the documents store since it is bound by the defining of columns for each record. Therefore, we decided to use the document store for implementing and testing MarketMaker data storage. Among many documents store, we found that MongoDB is a widely used, open-source, easy to set up distributed system for scalability, and abundant resource for development. Therefore, we investigated MongoDB technology with MarketMaker data.

4 Investigating MongoDB

MongoDB has been chosen for testing feasibility in migrating MarketMaker's database system because of its dynamic schemas using the JSON-based documents, its ability to handle a large amount of data with a fast search speed and horizontal scalability, and mainly, because of its simple and flexible development process.

4.1 Basic Setup

A MongoDB database for testing has been created using the current MarketMaker's dataset. In order to retrieve the data from the current MarketMaker database, we utilized the RESTful web services[11] developed for internal purposes. Using the RESTful web services, the current MarketMaker data has been transferred to the testing MongoDB server. The end results from the rest service were converted to JSON using the python script then inserted to MongoDB. During this process, the geospatial information is inserted using the original data's latitude and longitude information. In most data tables, data structures were acceptable for the direct transfer to MongoDB. However, the business data had a very complex data structure, so the special rest service and scripting for rearranging the business information from the MarketMaker's data structure to MongoDB's JSON format implemented for better query mechanism.

4.2 Migrating Data

The Python script controls the migration process. The script requests the data from RESTful service, and then it stores the data in MongoDB. During this process, insertion of the geospatial values to each JSON record is performed by converting the existing latitude and latitude data to a geospatial JSON object to enable the geospatial query. The geospatial object looks like the following:

```
{
   "loc":{
      "lon":[longitude value],
      "lat":[lattitude value]
   }
}
```

The dataset also has to be indexed using this ***loc*** field. After the insertion, a document looks like the following in MongoDB:

```
{
   "businesses":{
      "address":"12007 Highway 71",
      "state":"Illinois",
      "city":"Granville",
      "zipcode":"61326",
      "website":"none",
      "name":"Boggio's Orchard & Produce",
      "id":924,
      "registered":true,
      "phone":"815-339-2245",
      "loc":{
         "lat":41.257331,
         "lon":-89.240373
```

[11] http://en.wikipedia.org/wiki/Representational_state_transfer

```
        },
        "dist":-1,
        "businessTypes":[
            {
                "name":"Farmer",
                "id":2226
            },
            {
                "name":"Agritourism",
                "id":22390
            }
        ]
    }
}
```

4.3 Querying Data

The query test has been performed 1) using MongoDB console interface, and 2) using Google Web Toolkit (GWT). Note that GWT is used for current MarketMaker web application. The following sections show the example queries against the test MongoDB.

4.3.1 Using MongoDB console interface

The query shown below is finding a first record in the DB named, "test_coll".

```
db.test_coll.findOne()
```

MongoDB also allow users to perform a geospatial query by using a point with a certain radius and using a polygon. The query below is finding the first two records that lie within 50 miles radius from the location (51,-114) in the DB named, "test_coll":

```
db.test_coll.find({loc:{$near:[51,-114], $maxDistance:50}}).limit(2)
```

It is also possible to find all records that lie within the given polygon defined by two points, [[40.73083, -73.99756], [40.741404, -73.988135]], in the DB named, "places":

```
box = [[40.73083, -73.99756], [40.741404,  -73.988135]]
db.places.find({"loc" : {"$within" : {"$box" : box}}})
```

4.3.2 Using GWT (Java)

The following Java code is finding a record with business id, 4454. Note that the following code is the part of the Java code we developed.

```
searchQuery.put("id", 4454);
cursor = collection.find(searchQuery);
```

The code below is finding the first five records that lie within a 100 mile radius from the location (51,-104):

```
BasicDBObject filter = new BasicDBObject("$near", new double[] {55.0, -
104.0});
filter.put("$maxDistance", 100);
searchQuery.put("loc", new BasicDBObject(filter));
cursor = collection.find(searchQuery).limit(5);
```

It's also possible to find all records that lie within the given polygon defined by two points, [[34.94, -105.29], [[42.56, -87.65]]:

```
LinkedList<double[]> box = new LinkedList<double[]>();
// Set the lower left point
box.addLast(new double[] {  34.94, -105.29 });
// Set the upper right point
box.addLast(new double[] { 42.56, -87.65 });
BasicDBObject filter = new BasicDBObject("$within", new BasicDBObject("$box",
box));
searchQuery.put("loc", new BasicDBObject(filter));
cursor = collection.find(searchQuery).limit(5);
```

4.4 Advantages

Our test of MongoDB yielded two main advantages:

- Geospatial Query: MongoDB has built-in capability of geospatial query. This is a very crucial capability for MarketMaker since all the businesses are geo-referenced and most of the queries have geo-spatial aspect. Since MongoDB provides this capability natively, the system does not need any additional plug-in or processes for geospatial query. This will improve the search performance and lessen the computing burden on the server.
- Ease of horizontal scalability: MongoDB supports automatic sharding, failover, and recovery. It allows to store in multiple nodes (multiple databases), easy replication for high availability, and failure tolerable system.

4.5 Problems

Despite of the advantages listed above, we found the MongoDB lacks necessary features. The main feature missing is a capability of keyword search. The keyword search that current MarketMaker users need is not a simple text search. MongoDB also needs to support a facet search which allows users to narrow down their search based on the attributes of the search results. The missing feature can be achieved by other new technology called "index service". Since MongoDB can store flexible schema of

the data, the data is ready to be indexed by the “index service” to support the search capability. This detailed discussion is found in the next Section.

5 Recommendation

5.1 Data Structure

Even though MongoDB provides MarketMaker with very good capabilities for substituting the current MS-SQL based data system, using it in the MarketMaker system still requires changes/modification to the data structure in the ‘business type’ detail. The current system has a rather complicated way of recording business details, due to the wide variety of product/business types and details.

Also, current RESTful service provides the data structured in a tree-like form. If you directly transferred the current business detail data to the MongoDB system via RESTful service, it would look like the sample below in MongoDB. Note that [Title] and [Folder] represents a node in a tree structure. If the node were a leaf, there would be no [folder].

```
[businessTypesDetail] => Array (
    [0] => Array (
        [folder] => Array (
            [0] => Array (
                [folder] => Array (
                    [0] => Array (
                        [folder] => Array (
                            [0] => Array (
                                [title] => Other%3A+Condiments%2FPickled+Items
                            )
                        )
                        [title] => Specialty+Product+Type
                    )
                    [1] => Array (
                        [folder] => Array (
                            [0] => Array (
                                [title] => Bottled%2C+Canned%2C+Condiments%2C+Prepared%2FPackaged
                            )
                        )
                        [title] => Product+Forms
                    )
                    [2] => Array (
                        [folder] => Array (
                            [0] => Array (
                                [title] => Delivery%2C+Farmers+Market%2C+Internet%2C+Wholesale
                            )
                        )
                        [title] => Methods+of+Sale
                    )
                )
                [title] => Specialty+Products
            )
        )
        [title] => Farmer%2FRancher
    )
    [1] => Array (
        [folder] => Array (
```

```
            [0] => Array (
                [folder] => Array (
                    [0] => Array (
                        [folder] => Array (
                            [0] => Array (
                                [title] => Delivery%2C+Direct%2C+Internet+Sales%2C+Mail+Order
                            )
                        )
                        [title] => Methods+of+Sale
                    )
                    [1] => Array (
                        [folder] => Array (
                            [0] => Array (
                                [title] =>
Annual+Sales+%28Less+than+%24500%2C000%29%2C+Number+of+Employees+%281+to+4%29
                            )
                        )
                        [title] => Business+Details
                    )
                )
                [title] => Business+Details
            )
            [1] => Array (
                [folder] => Array (
                    [0] => Array (
                        [folder] => Array (
                            [0] => Array (
                                [title] => Specialty+Foods%2C+Spices
                            )
                        )
                        [title] => Other+Specialty+Food+Store
                    )
                    [1] => Array (
                        [folder] => Array (
                            [0] => Array (
                                [title] => Canned
                            )
                        )
                        [title] => Product+Forms
                    )
                )
                [title] => Other+Specialty+Food+Store
            )
        )
        [title] => Food+Retailer
    )
)
```

This type of data structure hinders the development of a fast, efficient, query system. Therefore, the business detail should be more organized when converted to JSON for storing in MongoDB. The suggested structure of the business detail for Food Retailer in JSON format is following:

```
{
   "BusinessType":"Food Retailer",
   "BusinessDetail":[
      {
         "BusinessTitle":"Baked Goods Store",
```

```
         "ProductDetail":[
            {
               "ProductTypeTitle":"Backed Goods Store",
               "ProductType":"Baked Goods",
               "ProductAttribute":"Ethnic",
               "MethodsofSale":"Delivery, Direct, Internet Sales",
               "BusinessDetails":"Annual Sales (Less than $500,000) "
            }
         ]
      },
      {
         "BusinessTitle":"Other Specialty Food Store",
         "ProductDetail":[
            {
               "ProductTypeTitle":"Other Specialty Food Store",
               "ProductType":"Cofffee & Tea, Ice Cream, Spices",
               "ProductAttribute":"Ethnic, Organic",
               "MethodsofSale":"Delivery, Direct, Internet Sales",
               "Business Details":"Annual Sales (Less than $500,000)"
            }
         ]
      }
   ]
}
```

The suggested structure of the business detail for Farmer/Rancher in JSON format is following:

```
{
   "BusinessType":"Farmer/Rancher",
   "BusinessDetail":[
      {
         "BusinessTitle":"Fruits ",
         "ProductDetail":{
            "ProductType":"Mayhaw Apples",
            "ProductAttribute":"CalciumFortified, No Presservatives",
            "ProductForms":"Bottled, Canned, Condiments, Prepared/Packaged",
            "MethodsofSale":"Delivery, Farmers Market, Internet, Wholesale",
            "MarketsServed":"Local,Noortheast(USA)",
            "BusinessDetails":"7AM - 5PM Monday through Friday"
         }
      }
   ]
}
```

If JSON data had a structure like shown above, the query in MongoDB would be much more organized and would enable various types of searches. Also, MarketMaker's web user interface for the new business registration should make the JSON data in this structure, and then insert into MongoDB.

As for the migration of the current business detail data, the special code or script should be made for converting MarketMaker's data to the suggested JSON structure. Since there are several types of business categories, each category should have a very carefully designed transfer mechanism. For example, food retailers and farmers should have a different migration method because their different data attributes, but should be transformed to JSON that has a common structure for most types of businesses. For this, the test python script has been made and the actual code is included in Appendix B. This code transfers two sample types of data - farmer/rancher and food retailer - to common JSON structure allowing the query more widely accepted across the business types.

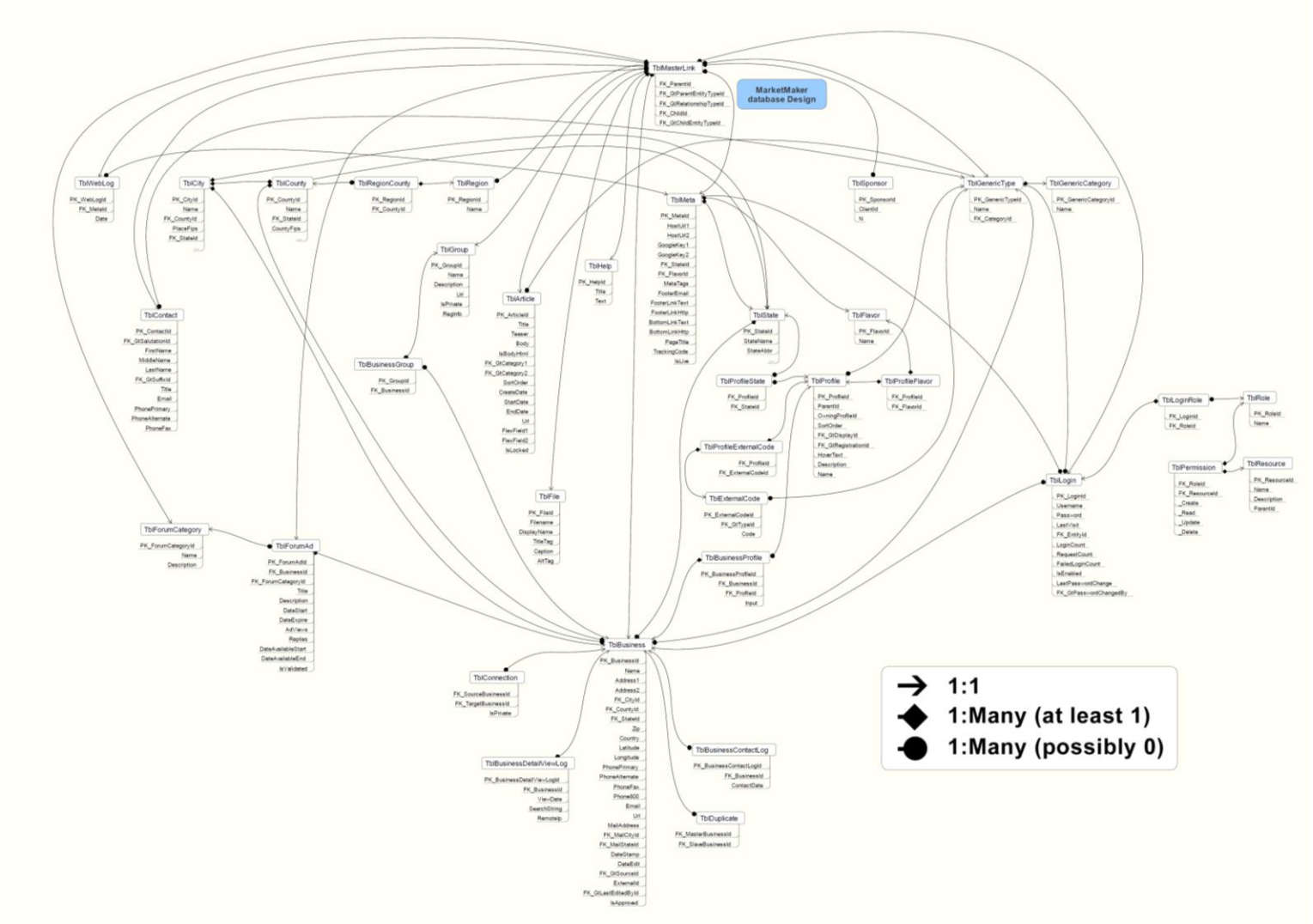
However, in order to make this change, we need to work with data experts in MarketMaker since reorganizing data model requires the domain knowledge of MarketMaker data.

5.2 Index Service

Along with the changes of MarketMaker's data structure, the data needs to be indexed with Index Service in order to support advanced search capabilities such as keyword search and facet search. We recommend Elasticsearch (http://www.elasticsearch.org/) as the index service for MarketMaker. There are two main reasons for this recommendation:

- Horizontal Scalability: the Elasticsearch project provides an easy way to build multiple nodes to store and index the documents like MongoDB. Moreover, it supports the near real-time indexing service when there is new data in the system.
- JSON document: the document in the Elasticsearch is in JSON format which is same as MongoDB. This means that it's easy to make documents to be indexed in Elasticsearch. Note that the document in MongoDB is full data while the document in Elasticsearch is subset of the full data.
- RESTful service: the Elasticsearch is accessed via RESTful service. It allows MarketMaker to be in SOA (service oriented architecture). SOA will make the MarketMaker system more reliable.
- Open source: the project is in open source (Apace 2) and has a very active community.

Appendix A: Current MarketMaker Data Schema



Appendix B: Python Script for data migration to MongoDB

```
import urllib2
import urllib
import re
import json
import pymongo
from pymongo import GEO2D

def main():
    url_base = "http://webapps.marketmaker.uiuc.edu/mobile/rest"
    url_middle = "/businesses"
    url_end = "/byproduct?productid=2268&lat=40.113789&lon=-
88.242187&maxdistance=500&page=1&limit=50&sortby=distance"

    """
    # test using the static url
    #url_final = url_base + url_middle + url_end
    # errored one
    #url_final =
"http://webapps.marketmaker.uiuc.edu/mobile/rest/businesses/5480/detail?type=
15035"
    #url_final =
"http://webapps.marketmaker.uiuc.edu/mobile/rest/businesses/5480/detail?type=
2226"
    url_final =
"http://webapps.marketmaker.uiuc.edu/mobile/rest/businesses/924/detail2?type=
2226"
    #url_final =
"http://webapps.marketmaker.uiuc.edu/mobile/rest/businesses/5480/detail2?type
=15035"
    outJson = getJsonHttp(url_final)
    insertSingleToMongo(outJson)
    return
    idList = parseIdFromJson(outJson)
    finalList = finalizeJson(idList, url_base, url_middle)
    insertToMongo(finalList)
    """

    url_final = urlMaker(url_base, url_middle)

    for i in range(len(url_final)):
        print "Processing " + str(i) + "th url......" + str(len(url_final) -
i) + " remains..."
        outJson = getJsonHttp(url_final[i])
        idList = parseIdFromJson(outJson)
        print str(len(idList)) + " records in this routine"
        finalList = finalizeJson(idList, url_base, url_middle)
        #insertToMongo(finalList)
        break

    #typeList = parseBusinessTypeId(idList, url_base, url_middle)

    #insertToMongo(outJson)
```

```
def mongoInsertTest(idList, url_base, url_middle):
    """
    get the business info with id
    for getting information in page 21
    """
    typeIdList = []

    connection = pymongo.Connection("localhost", 27017)
    db = connection.mm_testdb
    coll = db.mm_testcoll

    for i in range(len(idList)):
        url_final = url_base + url_middle + "/" + idList[i] + "/info"
        outJson = getJsonHttp(url_final)
        coll.insert(json.loads(outJson))


def urlMaker(url_base, url_middle):
    """
    generate url with the combination of busines product and location

    product id json
    [{"name":"Food Retailer","id":15035},
    {"name":"Buyer","id":23553},
    {"name":"Fishery","id":22695},
    {"name":"Farmer/Rancher","id":2226},
    {"name":"Winery","id":1742},
    {"name":"Wholesaler","id":20357},
    {"name":"Agritourism","id":22390},
    {"name":"Processor","id":19069},
    {"name":"Farmers Market","id":1734},
    {"name":"Eating & Drinking Place","id":15926}]

    Lat/Lon for the center-like point
    Colorado            -105.5506   38.9979
    Texas               -98.5150    31.1689
    Nebraska            -99.6809    41.5008
    Wyoming       -107.5546   43.0002
    Mississippi         -89.8764    32.5711
    New York            -75.7700    42.7466
    Illinois            -88.8326    39.9129
    Ohio                -82.6694    40.3653
    Kentucky            -84.5532    37.5621
    Michigan            -84.9691    43.0921
    Indiana       -86.4412    39.7665
    Pennsylvania        -77.6047    41.1179
    South Carolina      -80.9266    33.6245
    Alabama       -86.6807    32.5762
    Florida        -81.5579   27.6986
    Georgia       -83.1783    32.6782
    Iowa                -93.3898    41.9383
    Louisiana           -91.0538    29.9828
    Arkansas            -92.4782    34.6652
    """
```

```
    url_final = []
    url_end_location = ""
    url_end_product = ""

    productId = [15035, 23553, 22695, 2226, 1742, 20357, 22390, 19069, 1734,
15926]
    latLon = [(-105.5506, 38.9979), (-98.5150, 31.1689), (-99.6809, 41.5008),
(-107.5546, 43.0002), (-89.8764, 32.5711),
              (-75.7700, 42.7466), (-88.8326, 39.9129), (-82.6694, 40.3653),
(-84.5532, 37.5621), (-84.9691, 43.0921),
              (-86.4412, 39.7665), (-77.6047, 41.1179), (-80.9266, 33.6245),
(-86.6807, 32.5762),(-81.5579, 27.6986),
              (-83.1783, 32.6782), (-93.3898, 41.9383), (-91.0538, 29.9828),
(-92.4782, 34.6652)]

    for i in range(len(productId)):
        url_end_product = "/byproduct?productid=" + str(productId[i])
        for j in range(len(latLon)):
            url_end_location = "&lat=" + str(latLon[j][1]) + "&lon=" +
str(latLon[j][0]) + "&maxdistance=50&page=1&limit=50000&sortby=distance"
            url_final.append(url_base + url_middle + url_end_product +
url_end_location)

    return url_final


def parseBusinessTypeId(idList, url_base, url_middle):
    """
    get the business info with id
    for getting information in page 21
    """
    typeIdList = []

    for i in range(len(idList)):
        url_final = url_base + url_middle + "/" + idList[i] + "/info"
        outJson = getJsonHttp(url_final)
        x = json.loads(outJson)
        x1 = x['businessTypes']
        for j in range(len(x1)):
            typeIdList.append(str(x1[j]['id']))
    return typeIdList


def insertToMongo(jsonList):
    connection = pymongo.Connection("localhost", 27017)
    db = connection.test_db2
    coll = db.test_coll2

    #post = {"author":"Mike"}
    #post_json = json.dumps(post)
    #post_json_c = post_json.replace("\"","")

    #collection.insert(post)
    for i in range(len(jsonList)):
        #coll.insert(json.loads(jsonList[i]))
        coll.insert(jsonList[i])
```

```
    #ensure location index
    coll.ensure_index([("loc", GEO2D)])


def insertSingleToMongo(outJson):
    connection = pymongo.Connection("localhost", 27017)
    db = connection.test_db3
    coll = db.test_coll3

    #post = {"author":"Mike"}
    #post_json = json.dumps(post)
    #post_json_c = post_json.replace("\"","")
    tmpJson = json.loads(outJson)

    #coll.insert(tmpJson)


def finalizeJson(idList, url_base, url_middle):
    """
    parse lat/lon information and regenerate the geospatial json
    """
    spatialJsonList = []
    print "finalizing json list....."

    for i in range(len(idList)):
        url_final = url_base + url_middle + "/" + idList[i] + "/info"
        outJson = getJsonHttp(url_final)
        tmpJsonList = []

        # build up spatial info
        x = json.loads(outJson)
        lat = x['lat']
        lon = x['lon']
        x.pop("lat")
        x.pop("lon")
        #locJson = {"lon": lon, "lat": lat}
        x.update({"loc": {"lon": lon, "lat": lat}})

        # build up business type id
        x1 = x['businessTypes']
        for j in range(len(x1)):
            url_busType = url_base + url_middle + "/" + idList[i] +
"/detail2?type=" + str(x1[j]['id'])
            tmpJson = reconstructBusJson(url_busType, str(x1[j]['id']))
            if tmpJson != None:
                tmpJsonList.append(json.loads(tmpJson))
            break
        x.update({"businessTypesDetail":tmpJsonList})
        spatialJsonList.append(x)
        break

    return spatialJsonList


def reconstructBusJson(url_busType, bus_id):
    """
    """
```

```
    #tmpJson.append(json.loads(getJsonHttp(url_busType)))
    x = json.loads(getJsonHttp(url_busType))
    #x = getJsonHttp(url_busType)

    # farmer/rancher
    if bus_id == "2226":
        tmpJson = "{\"BusinessType\":"
        tmpJson = tmpJson + "\"" + x["title"] + "\",\"BusinessDetail\":["
         # specialty products

         #Other condiments..
        for i in range(len(x["folder"])):
            if i == 0:
                tmpJson = tmpJson + "{\"BusinessTitle\":\"" +
x["folder"][i]["title"] + "\",\"ProductDetail\":["
                print x["folder"][i]["title"]
            else:
                tmpJson = tmpJson + ",{\"BusinessTitle\":\"" +
x["folder"][i]["title"] + "\",\"ProductDetail\":["
                print x["folder"][i]["title"]
            for j in range(len(x["folder"][i]["folder"])):
                if j == 0:
                    tmpJson = tmpJson + "{\"" +
x["folder"][i]["folder"][j]["title"] + "\":"
                    print x["folder"][i]["folder"][j]["title"]
                else:
                    tmpJson = tmpJson + ",{\"" +
x["folder"][i]["folder"][j]["title"] + "\":"
                    print x["folder"][i]["folder"][j]["title"]
                for k in range(len(x["folder"][i]["folder"][j]["folder"])):
                    tmpJson = tmpJson + "\"" +
x["folder"][i]["folder"][j]["folder"][k]["title"] + "\"}"
                    print x["folder"][i]["folder"][j]["folder"][k]["title"]
            tmpJson = tmpJson + "]}"
        tmpJson = tmpJson + "]}"
        #print tmpJson

        return tmpJson

    # food/retailer
    elif bus_id == "15035":
        tmpJson = "{\"BusinessType\":"
        tmpJson = tmpJson + "\"" + x["title"] + "\",\"BusinessDetail\":["
        for i in range(len(x["folder"])):
            if i == 0:
                tmpJson = tmpJson + "{\"BusinessTitle\":\"" +
x["folder"][i]["title"] + "\",\"ProductDetail\":["
            else:
                tmpJson = tmpJson + ",{\"BusinessTitle\":\"" +
x["folder"][i]["title"] + "\",\"ProductDetail\":["
            for j in range(len(x["folder"][i]["folder"])):
                if j == 0:
                    tmpJson = tmpJson + "{\"" +
x["folder"][i]["folder"][j]["title"] + "\":"
                else:
                    tmpJson = tmpJson + ",{\"" +
x["folder"][i]["folder"][j]["title"] + "\":"
```

```
                for k in range(len(x["folder"][i]["folder"][j]["folder"])):
                    tmpJson = tmpJson + "\"" +
x["folder"][i]["folder"][j]["folder"][k]["title"] + "\"}"
            tmpJson = tmpJson + "]}"
        tmpJson = tmpJson + "]}"

        return tmpJson


def parseIdFromJson(outJson):
    """
    parse id in page 16 from the returned json
    and make it as a list
    """
    print "parsing id from json...."
    idList = []

    x = json.loads(outJson)

    for i in range(len(x)):
        idList.append(str(x[i]['id']))

    return idList


def getJsonHttp(url):
    # one way
    req = urllib2.Request(url)
    opener = urllib2.build_opener()
    f = opener.open(req)
    outJson = f.read()
    outJson1 = urllib.unquote_plus(outJson)

    return outJson1

    # the other way
    #data = urllib2.urlopen(url_final)
    #data = re.compile('^[^\(]+\(|\)$').sub('', data.read())
    #parseData = json.loads(data)


main()
```